CS2040s Cheatsheet (Compact)

github.com/reidenong/cheatsheets, AY23/24 S2

Time Complexity

$T(n) = O(f(n))$ if $\exists c, n_0 > 0$ s.t. $\forall n > n_0, T(n) \leq cf(n)$
$T(n) = \Omega(f(n))$ if $\exists c, n_0 > 0$ s.t. $\forall n > n_0, T(n) \geq cf(n)$
$T(n) = \Theta(f(n)) \Leftrightarrow T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

Order of Growth

$O(1) \to O(\log \log N) \to O(\log N) \to O(\log^2 N) \to O(\sqrt{N}) \to O(N) \to O(N \log N) \to O(N^2) \to O(2^N) \to O(2^{2N}) \to O(N!)$

Miscellaneous Identities

- $\sum_i^N \frac{1}{i} = O(\log N)$
- $T(n) = T(n-1) + T(n-2) + O(1) = O(2^n)$

Searching

Binary Search: $O(\log N)$

```
int binarySearch(int[] arr, int x)
  int lo = 0, hi = arr.length - 1;
  while (lo < hi)
      int mid = lo + (hi - lo)/2;
      if (in lower half)
          end = mid;
      else
          start = mid + 1;
  return lo;
```

Kth smallest element (Quickselect): $O(N)$
- DnC, partition array into 2 halves, recurse on the half that contains the kth element

DnC Peak Finding: $O(\log N)$
- Operates on same concept as binary Search, DnC

(N x M) 2D Peak Finding: $O(N + M)$
- Quadrant Divide and Conquer

Sorting

Bubble Sort
- Swap every inversion pair a single pass. Repeat N times.
- Inv: At the end of iteration j, the last j elements are in their correct position, **ie. Globally sorted suffix**
- Best case: $O(N)$, Worst case: $O(N^2)$
- In-place and stable

Selection Sort
- Repeatedly extract min element from the unsorted suffix.
- Inv: At the end of iteration j, the first j elements are in their correct position, **ie. Globally sorted prefix**
- Best case / Worst case: $O(N^2)$
- In-place but unstable

Insertion Sort
- Repeatedly insert items from unsorted suffix into sorted prefix.
- Inv: At the end of iteration j, the first j elements are sorted locally, **ie. Locally sorted prefix**
- Very fast on almost-sorted arrays
- Best case: $O(N)$, Worst case: $O(N^2)$
- Average case: $\sum_{j=2}^{N} \Theta\left(\frac{j}{2}\right) = \Theta(N^2)$
- In-place and stable

Merge Sort
- Inv: **Locally sorted prefixes in powers of two**
- Best case / Worst case: $O(N \log N)$
- $O(N \log N)$ space and stable.

* May perform slower for small $N (< 1024)$ due to cache performance, branch prediction, general overhead costs

QuickSort (Hoare partitioning)
- Inv: $\forall i, arr[i] \leq x$ for $i < p$ and $arr[i] > x$ for $i > p$

  ie. Sorted around pivots, which are in position
- Optimizations: Randomized pivot, 3-way partitioning, insertion sort for small N
- Best case: $O(N \log N)$, Worst case: $O(N^2)$
- In-place but unstable
- Paranoid QuickSort: Expected time is $O(N \log N)$

Counting Sort
- Determine idx $\forall$ keys by counting number of objects with distinct key values and applying prefix sums.
- Time: $O(N + k)$ where k is the range of the input
- $O(N + k)$ space and stable

Radix Sort
- Sort the input numbers by their individual digits.

Time: $O(Nd)$ where d is the number of digits
Space: $O(N + k)$
Stable: Yes

Heap Sort
- Heapify in $O(N)$, extractMax() N times
- Faster than MergeSort, slower than Quicksort
- Best Case / Worst Case: $O(N \log N)$
- In-place but unstable

Trees

Binary Search Tree:
- $O(h)$ / $O(N)$: insert, delete, predecessor, successor, search, findMax, findMin
- **Strictly** $O(N)$: Traversal

bBST / AVL Trees:

A BST is balanced if $h = O(\log N)$. A node is height-balanced if the height of its left and right subtrees differ by at most 1. A tree is height-balanced if all its nodes are height-balanced.

A height balanced tree with N nodes has at most height $h < 2 \log N \leftrightarrow$ A height balanced tree with height h has at least $N > 2^{\frac{h}{2}}$ nodes.

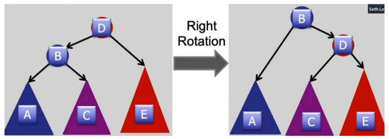
Upper bound of nodes in a AVL tree
$N_h \leq 1 + 2N_{h-1} \leq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$
Lower bound of nodes in a AVL tree:
$N_h \geq 1 + N_{h-1} + N_{h-2} \geq 2N_{h-2} = 2^{\frac{h}{2}}$

Operations

Insertion:
- Insert node at leaf, then walk up tree, only need to fix lowest unbalanced node, maximum 2 rotations

Deletion:
- If not leaf, swap with successor. Delete. Fix all unbalanced nodes until root, up to $O(\log N)$ rotations

Rotations




```
def leftRotate(B) :
  D = B.right
  D.parent = B.parent
  B.parent = D
  B.right = D.left
  D.left = B
  return D
```

```
def rightRotate(D) :
  B = D.left
  B.parent = D.parent
  D.parent = b
  D.left = B.right
  B.right = D
  return B
```

Balancing AVL Trees

WLOG, a node v is **left heavy** if left subtree has larger height than right subtree

If v is left heavy :
(1) v.left is balanced or left heavy : rightRotate(v)
(2) v.left is right heavy : leftRotate(v.left), rightRotate(v)

If v is right heavy :
(1) v.right is balanced or right heavy : leftRotate(v)
(2) v.right is left heavy : rightRotate(v.right), leftRotate(v)

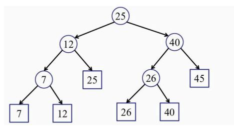
Trie:
- Independent of number of elements, operations are $O(L)$ where L is the length of the key.
- Faster than the $O(Lh)$ alternative of strings in a bBST, though it has more nodes and thus more overhead space

Order statistics: $O(\log N)$
- AVL tree, augment each node with the size of its subtree.
- select(i) : find the ith smallest element
- rank(x) : find the rank of element x

Interval Queries: $O(\log N)$
- Sort all intervals by left endpoint in bBST
- $\forall$ nodes store the maximum right endpoint in the subtree rooted at that node

(Dynamic) 1D Range Queries

Finding all elements in a range $[a, b]$.



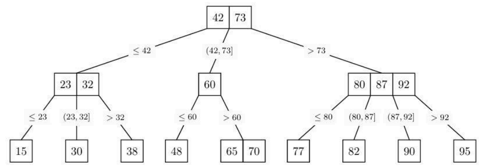
- All elements are leaves. Each internal node stores the maximum value in its left subtree
- (1) Find split node $O(\log N)$, then (2) Do left and right traversals
- Inv: Search interval for a left-traversal at node v includes the maximum item in the in the subtree rooted at v
- Preprocessing: $O(N \log N)$ for N insertions of $O(\log N)$
- Query: $O(\log N + k)$ where k is the number of elements in the range
- Tree can be augmented with the count of each node in subtree to support counting queries in $O(\log N)$
- Space: $O(N)$

(Static) 2D Range Queries:
- Build a 1D range tree for all x-coords, for each internal node, build a y-tree for all y-coords
- Preprocessing: $O(N \log N)$
- Query: $O(\log N)$ to find split node, $O(\log N)$ recursing steps, $O(\log N)$ y-tree searches each of $O(\log N)$, $O(k)$ enumerating output. **Total:** $O(\log^2 N + k)$
- Space: $O(N \log N)$
- Not dynamic: $O(N)$ for rebuilding y-trees if rotate

D - dimension Range Queries:

In general for a $d-$ dimension range query,
- Query cost: $O(\log^d N + k)$
- Preprocessing: $O(N \log^{d-1} N)$
- Space: $O(N \log^{d-1} N)$

(a, b)-Trees



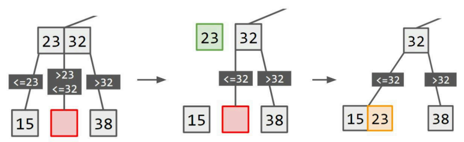
- Nodes have at least a and at most b children, $2 \leq a \leq \frac{b+1}{2}$.
- v_1 key range $\leq v_1$, v_k key range $> v_k$, else $(v_{i-1}, v_i]$
- All leaf nodes have same depth, tree grows upwards.
- **Search:** $O(b \log_a N) = O(\log N)$ for N elements.

(Proactive) Insertion: $O(\log N)$
- Insertion may cause a node to have too many keys. We preemptively split full nodes (ie. $b-1$ keys), guaranteeing parent of the node to be split will not have too many keys after insertion of split keys

Split: $O(b)$ for splitting a node with b children
- When a node u has b children and a new child is added, offload median node to parent, and split the remaining children into 2 nodes
- If we are splitting z, then preconditions are
1. z has $\geq 2a$ keys. After splitting and offering a key to parent, LHS has $\geq a-1$ keys and RHS has $\geq a$ keys
2. z's parent has $\leq b-2$ keys.

(Lazy) Deletion: $O(\log N)$
- First delete target key, then recursively check upwards for violation with carrying out merge/share operations. For internal, replace with predecessor/successor. Suppose we are deleting a key from node z, which also has smallest sibling y. Deletion risks having nodes shrink to small. Given that z has $< a-1$ keys,

If z **and** y **have** $< b-1$ **keys together:** `merge(y, z)`



If z **and** y **have** $\geq b-1$ **keys together:** `share(y, z)`
- `merge(y, z)` gives node w with $\geq b$ keys, `split(w)` gives us nodes y and z with $\geq a-1$ keys

Hashing (n items, m buckets)

Chain Hashing

Simple Uniform Hashing Assumption:

- Each key is equally likely to map to any bucket, and keys are mapped independently to buckets
- Expected search time = $O(1) + \alpha = O(1)$
- Worst case search time = $O(n)$
- Worst case Insertion time = $O(1)$
- Expected Maximum chain length = $\Theta\left(\frac{\log n}{\log \log n}\right) = O(\log n)$

Hashing Table Sizing

Assuming hashing with chaining with SUHA,

- Optimal size = $m = \Theta(n)$. If $m < 2n$, there are too many collisions; If $m > 10n$, there is too much wasted space.

Growing the current m_1 table:
(1) Choose new table size m_2. (2) Choose new hash function h based on table size. (3) Rehash all keys.

- Scanning old table: $O(m_1)$
- Creating new table: $O(m_2)$
- Inserting each of n keys: O(1)
- Total: $O(m_1 + m_2 + n) = O(n)$

Deleting elements: $O(1 + \alpha)$

- We calculate the key's hash, then search in it's hash bucket.

Amortized Analysis

- Operation has amortized cost $T(n)$ if for every integer k, the cost of k operations is $\leq kT(n)$

Shrink/Grow policy

if $n == m$, then $m = 2m$
if $n < \frac{m}{4}$, then $m = \frac{m}{2}$

- If we double a table of size m, there must have been at least $\frac{m}{2} = O(m)$ insertion operations to spread cost over
- If we halve a table of size m, there must have been at least $\frac{m}{2} = O(m)$ deletion operations to spread cost over
- Operations remain $O(1)$ amortized

Open Addressing

On collision, we probe a sequence of buckets until we find an empty slot, $h(k, i) \equiv$ `h(key, numOfCollisions)`

Good hashing Properties

- $h(k, i)$ enumerates all possible buckets, ie. the hash function is some permutation of the buckets.
- Uniform Hashing Assumption: Each key is equally likely to be mapped to every permutation, independent of every other key

Linear Probing:

- Does not satisfy UHA, eg. permutations such as 1 2 4 3 will never appear. Clusters tend to develop; If table $\frac{1}{4}$ full, there will be clusters of size $\Theta(\log N)$
- However, it may be faster due to caching, as it is cheap to access nearby array cells, eg. if the cache holds the entire cluster

Double Hashing:

- $h(k, i) = (f(k) + i \cdot g(k)) \bmod m$ with hash functions f, g
- if $g(k)$ relatively prime to m, then h hits all buckets.

Performance of Open Addressing with UHA

Expected operation time
$= 1 + \frac{n}{m}\left(1 + \frac{n-1}{m-1} + \left(1 + \frac{n-2}{m-2}(\text{Cost of Remaining probes})\right)\right)$
$\approx 1 + \alpha(1 + \alpha(1 + \alpha(\ldots))) = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots = \frac{1}{1-\alpha}$

Pros and Cons of Open Addressing

Pros: Saves space (no linked lists); Rarely allocates memory (no list-node allocations); Better cache performance

Cons: More sensitive to hash function quality, eg. clustering, linear probing; More sensitive to load factor, runtime increases exponentially as α approaches 1

Hash Set (Fingerprint Hash Table)

- Stores 0/1 bool for each key without value to reduce space
- If item is in set, then it will always return true (No false negatives). If item is not in set, it may return true (False positives).

Probability of **no false positives** (under SUHA)
= probability of no collision for each of n items
$= \left(1 - \frac{1}{m}\right)^n \approx \left(\frac{1}{e}\right)^{\frac{n}{m}}$
To have probability of false positives < p, $\frac{n}{m} \leq \log\left(\frac{1}{1-p}\right)$

Binary Heaps (MaxHeap)

- Complete binary tree, every level is full except possibly the last, all nodes are as far left as possible. priority[parent] >= priority[child]. Max height = $\lfloor \log N \rfloor = O(\log N)$

Insertion: $O(\log N)$

- Insert x to leaf node, then bubble up

Delete / Extract Max: $O(\log N)$

- Replace x / Max element with last node, then bubble down

increaseKey / decreaseKey: $O(\log N)$

- Increase / decrease key, then bubble up / down

Heap vs AVL:

- Same asymptotic cost
- Heap has faster real costs
- Heap is simpler with no rotations
- Heap has better concurrency

Array implementation:

- Parent of node i is at index $\left\lfloor \frac{i-1}{2} \right\rfloor$
- Left child of node i is at index $2i + 1$
- Right child of node i is at index $2i + 2$

Heapify: $O(N)$

- Base case: Leaf nodes are already heaps
- Recursive step: $\forall$ nodes, if their children are heaps, add that node to the heap and `bubbleDown`.

$\sum_{h=0}^{\log N} \frac{N \cdot O(h)}{2^h} \leq cN\left(\frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \ldots\right) \leq cN\left(\frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2}\right) = O(N)$

Priority Queues

- Use Binary heap / BBST + Hash Table
- Insert in both BBST + Hash Table, $O(\log n)$
- Delete in both BBST + Hash Table, $O(\log n)$
- (contains) Search in Hash Table, $O(1)$

Graphs

Terminology

- Path: Set of edges connecting two nodes
- Connected: Every pair of nodes is connected by a path
- Degree: Number of edges connected to a node
- Diameter: Max shortest path distance between two nodes
- Clique: Complete Graph
- Bipartite: Nodes can be divided into 2 sets, with no edges within a set
- Dense: $|E| = \Theta(V^2)$

Diameter of the graph of a $n \times n \times n$ rubik's cube = $\Theta\left(\frac{n^2}{\log n}\right)$

	adjMatrix	adjList
Space	$O(V^2)$	$O(V + E)$
Cycle	$O(V^2)$	$O(V)$
Clique/Complete	$O(V^2)$	$O(V^2)$
Query neighbour(v, w)	Fast	Slow
Find any neighbour	Slow	Fast
Enumerate all neighbours	Slow	Fast

Shortest Paths

Triangle Inequality

For any 3 nodes, u, v, w, $\delta(u, w) \leq \delta(u, v) + \delta(v, w)$

Bellman-Ford: $O(VE)$

- Relax all edges $V - 1$ times
- Inv: Let P be a shortest path from s to v. After i iterations, if node u is i hops from s on P, `est[u]` is the length of the shortest path from s to u. (may not be all nodes i from s)
- Can stop after one iteration with no updates
- Detects negative cycles (runs $> n$ times)

Dijkstra: $O((V + E)\log V) = O(E \log V)$

DAG TopoSort

- **Post-Order DFS with prepend**: $O(V + E)$
- **Kahn's Algorithm:** $O(V + E)$

Condition	Algorithm
No Negative weight cycles	Bellman-Ford
No Negative Weights	Dijkstra
Unweighted Graph	BFS
On Tree	BFS/DFS
DAG	TopoSort

UFDS

- Quick Find: Flat Trees with arbitrary union
- Quick Union: Connect the roots of 2 trees arbitrarily
- WU: Connect the root of smaller tree to root of larger tree
- PC: on `find()`, set parent of all nodes on path to root

Maximum depth of UFDS tree

Induction that a tree of height k has at least 2^k nodes

1. Assume T_1 has height $k - 1$ and is made the child of another tree.
2. T_2 has size $\geq 2^{k-1}$
3. size(T_2) $\geq$ size(T_1) $\geq 2^{k-1}$
4. size($T_1 + T_2$) $\geq 2^k$
5. Tree of height k has $\geq 2^k$ nodes
6. Height of tree of size n is $\leq \log n$

	find	union
quick find	$O(1)$	$O(N)$
quick union	$O(N)$	$O(N)$
weighted union	$O(\log N)$	$O(\log N)$
path compression	$O(\log N)$	$O(\log N)$
WU + PC for m operations	$a(m, n)$	$a(m, n)$

MSTs

Properties of MSTs

1. No cycles
2. Every cut of a MST produces two MSTs
3. For every cycle, the max edge weight is not in the MST
4. For every partition of the MST, the minimum edge weight across the cut is in the MST

Generic MST Algorithm

- For each cycle with no red edges, color the max edge red
- If D is a cut with no blue arcs, color the minimum edge blue
- Greedily apply red/blue rule until no more edges can be colored, blue edges form an MST

Prim's Algorithm: $O(E \log V)$ with priority queue

- Add node to the MST set $\rightarrow$ add all edges from the new node to nodes not in the MST set
- Pick the min edge and add it's node to MST set (blue rule)

Kruskal's Algorithm: $O(E \log V)$ with UFDS

- Sort all edges by weight. For each edge, either discard it if it forms a cycle (red rule) or add it if it does not (blue rule)

Directed Graph (with root): $O(E)$

- For each node except root add minimum incoming edge

Dynamic Programming

LIS (from the right), $O(N^2)$
Sub-problem:

S[i] = LIS(A[i,...,n] starting at A[i])

Recurrence:

S[i] = max(S[i] if A[j] > A[i] fo r j = i+1,...,n) + 1

Prize Collecting on Directed Graph (in k steps), $O(kE)$
Sub-problem:

P[v][k] = Maximum prize collected starting at v in k steps

Recurrence:

P[v][k] = max(P[u][k-1] + prize(u, v) for u in incoming(v))

Vertex Cover on a Tree, $O(V)$
Sub-problem:

S[v][0] = size of vertex cover of subtree rooted at v, when v NOT covered,
S[v][1] = size of vertex cover of subtree rooted at v, when v is covered,

Recurrence:

S[v][0] = sum(S[w][1] for w in v.children)
S[v][1] = 1 + sum(min(S[w][0], S[w][1]) for w in v.children)

APSP, Floyd-Warshall, $O(V^3)$
Sub-problem:

S[v][w][P] = shortest path from v to w using only nodes in set P

Recurrence:

S[v][w][P] = min(S[v][w][P-{k}], S[v][k][P-{k}] + S[k][w][P-{k}])